Using Genetic Algorithm for Parameter Estimation

Yi Wang
Computer Science Department, Tsinghua University, 100084, Beijing, China
wangy01@mails.tsinghua.edu.cn

September 30, 2004

Abstract
This is a learning note of genetic algorithm.

Contents

1 Introduction ............................................. 2
1.1 The Problem and Concepts .......................... 2

2 Estimate Means of Mixture Distribution .......... 2
2.1 The Model ........................................ 3
2.2 Drawing Samples .................................. 3
2.3 Visualize the Problem Domain .................... 3
2.4 Search for Peak with GA ......................... 4

A APPENDIX ........................................... 5
A.1 MATLAB Programs in This Experiment ......... 5
A.1.1 The 1D Gaussian function ...................... 5
A.1.2 Draw Samples from Gaussian distribution .... 6
A.1.3 Compute the Likelihood for Any Given Parameter 6
A.2 The GA Algorithm ................................ 7
A.2.1 The Source Code in C++ ...................... 7
1 Introduction

Genetic algorithm is a method of searching, search for a result equal to or close to the answer of a given problem. In the principle of artificial intelligence, the problem of training a generative mode given a set of data (sometimes called parameter estimation) is a typical searching problem — search for parameters to adjust the model that generates data the most likely to the given training data. Many solutions have been given for different models. In this note, we show how to estimate parameters of a mixture distribution with example and experiments.

1.1 The Problem and Concepts

It is called optimization that search for answers that best fit some requirements. There are many solutions to different types of optimization problems, such as conjugate gradient method, primal-dual method, simplex method and many many others.

A typical subset of the optimization problems is estimating parameters of a generative model given a set of data generated form the model. A Generative model is usually some statistical model that approximate certain data generation procedure (that is why it is called generative). Some parameters adjusts gritty details of the data generation process. The parameter estimation (or model training in AI term) is (1) given a set of data \(O\) generated from the model with parameters \(\Theta_o\), (2) to find parameters \(\Theta_e\) that most close to \(\Theta_o\), since we lost the \(\Theta_o\) or do not know it at all.

A famous method for estimate model parameters given a set of partial data or noisy data is maximization expectation algorithm (EM). A fact worthy noticing is: in many problems, why the data is partial is not because it is really partial, but because we try to model it as partial, so EM can be applied.

From another aspect, if we model the finite set of parameters as a chromosome string, i.e. each parameter corresponding to a gene in the chromosome. Usually, when the genes are discrete values, like integers, we can find the optimize result with genetic algorithm (GA); when the genes are continuous values, we can find result close to the real value.

2 Estimate Means of Mixture Distribution

In this section, we demonstrate using genetic algorithm for estimating parameters of a 2-component mixture Gaussian distribution. A mixture distribution with \(n\) components is,

\[
P(o) = P(o|\theta_1)P(\theta_1) + P(o|\theta_2)P(\theta_2) + \ldots + P(o|\theta_n)P(\theta_n)
= \sum_{i=1}^{n} P(o|\theta_i)P(\theta_i)
\] (1)

Which means, for a given observed sample \(o\), it might be drawn from the \(i\)th component \(P(o|\theta_i)\) with probability of \(P(\theta_i)\). The probability of observing \(o\) is the summation of possibility from all the possible components.\(^2\)

The steps of our experiment contains:

1. Select a model,

\(^1\)You can find in the next section for descriptions of most concepts mentioned in this section.

\(^2\)The problem can be solved with EM by considering the \(o\) is partial data, and the complete sample is \([o, i]\), where \(i\) is the index of component that generates \(o\).
2. Draw samples $O$ from the model with parameters of $\Theta_o$.
3. Now, we forget $\Theta_o$, and try to find a $\Theta_e$ that best explain the drawn samples $O$. In some cases, such $\Theta_e$ might close to $\Theta_o$, but this is not usually true.

2.1 The Model
In our experiment, the each component is 1D Gaussian, i.e.

$$P(o|\theta_i) = \mathcal{N}(o; \mu_i, \sigma_i)$$

(2)

In order to give an intuitive visualization of the problem domain in 3D space, we reduce the number of parameter to 2 by

- fixing prior probability to uniform distribution $P(\theta) = (0..1)$, that is, for $n$ components
  $$P(\theta_1) = P(\theta_2) = \ldots = P(\theta_n) = \frac{1}{n}$$

(3)

- fixing the number of components to 2, and
- fixing $\sigma_i$ of all components to a known number. So we simplify the Gaussian notation containing only the mean value,

$$P(o|\theta_i) = \mathcal{N}(o; \mu_i)$$

Thus, our mixture distribution can be written as

$$p(o) = \sum_{\theta} p(o, \theta)$$

$$= p(\theta_1)\mathcal{N}(o; \mu_1) + p(\theta_2)\mathcal{N}(o; \mu_2)$$

(4)

$$= 0.5\mathcal{N}(o; \mu_1) + 0.5\mathcal{N}(o; \mu_2)$$

As you see, there are only 2 parameter, $\mu_1$ and $\mu_2$ left.

2.2 Drawing Samples
Now we fix $\sigma$ to 0.3, $\mu_1$ to 1, and $\mu_2$ to $-1$. We draw 3 samples from the first components, and 3 from the second. The mixture distribution and the drawn samples are shown in figure 1.

2.3 Visualize the Problem Domain
With the model type (not the parameters) known, we can compute the likelihood between the samples $O = \{o_1, o_2, \ldots, o_6\}$ and a given parameter set $\{\theta_1, \theta_2\}$. Since each $o_i$ might be generated from either of the 2 components, so the component correspondence of the 6-tuple $O$, denoted as $\Theta = \{\theta_1, \theta_2, \ldots, \theta_6\}$, has $2^6 = 64$ possible combinatorial values, denoted as $\times$. The computation of likelihood is as by considering
the 6 generations are i.i.d., and the components are independent to each other,

\[
p(O) = \sum_{\Theta \in \times} p(O, \Theta) = \sum_{\Theta \in \times} p(O|\Theta)p(\Theta)
= \sum_{\Theta \in \times} \prod_{i=1}^{6} p(o_i|\theta_i) \prod_{i=1}^{6} p(\theta_i)
= \sum_{\Theta \in \times} \prod_{i=1}^{6} N(o_i; \theta_i) \prod_{i=1}^{6} 0.5
\]  

(5)

A brute force solution to our problem is to enumerate all possible parameter-pairs \(\{\theta_1, \theta_2\}\). Unfortunately this method is impractical, because the range of \(\theta\) is infinite. But it can be used to visualize the solution domain, as in figure 2. There are two points in this figure need to be noticed,

- There are two peaks on the likelihood figure. Because \(\{-1, 1\}\) and \(\{1, -1\}\) are equivalent.\(^3\)

- Physically, this figure is the likelihood defined on the drawn samples, but not the true-likelihood of the model. So the peaks are not on \((-1, 1)\) and \((1, -1)\) exactly. For example, in figure 2, one peak intersecting with the transparent plane is at \((0.86, -1.03)\).

### 2.4 Search for Peak with GA

From figure 2, it is clear that want we want is the position of one of the two peaks. If we solve this problem with other methods, such like EM algorithm, we will start from one position, and close to the peak with each iteration. With GA, we start from a population of starting positions — each individual in the population is represented with its chromosome — a string of genes. In our example, a chromosome is consist of two genes, one for \(\theta_1\) and the other for \(\theta_2\). Within each iteration, a fitness value is calculated

\(^3\)How many peaks will have for a 3-component mixture? Yeah, \(P_3^3 = 6\).
Figure 2: Visualization of likelihood between all possible parameter pairs and the 6 drawn samples for each individual in the population. In our example, we choose the likelihood value shown in figure 2 at the fitness value, and then genetic operations are applied to the current population:

- **elitism**: elites (with higher likelihood to the 6 drawn samples) are survived — copied to the population of the next generation,
- **mate**: some ratio of individual reproduce offspring by averaging the parents’ values,
- **mutate**: the children mutate by add a random number to the $\theta_1$ and/or $\theta_2$ value.

Figure 3 shows the likelihood figure together with result of each iteration of our GA algorithm. The red line segments shows the process of close to one of the peaks.

### A APPENDIX

#### A.1 MATLAB Programs in This Experiment

**A.1.1 The 1D Gaussian function**

```matlab
function y = gauss1d(x, mu, sigma2)
%% Vector version of 1D Gaussian distribution (normal distribution)
for i=1 : length(x)
    y(i) = gauss1d_s(x(i), mu, sigma2);
end

function y = gauss1d_s(x, mu, sigma2)
```

5
% Scalar version of 1D Gaussian distribution
k = 1/sqrt(2 * pi * sigma2);
e = - (x-mu)*(x-mu) / (2* sigma2);
y = k * exp(e);

A.1.2 Draw Samples from Gaussian distribution

function OSS = draw_gauss(N, mu, Sigma)
    OSS = randn(1,N) * sqrtm(Sigma) + repmat(mu,1,N);

A.1.3 Compute the Likelihood for Any Given Parameter

function L = true_likelihood(O, mu1, mu2, sigma)
% Matrix version of the true_likelihood between
% a set of points and a given parameter-pair

    L = ones(length(mu1), length(mu2));
    for i=1:length(mu1)
        for j=1:length(mu2)
            Mu = [ mu1(i), mu2(j) ];
            L(i,j) = true_likelihood_s(O, Mu, sigma);
        end;
    end;

function L = true_likelihood_s(O, Mu, sigma)
% Scalar version of the true_likelihood between
\% a set of points and a given parameter-pair

T = ones(size(O,1),size(O,2));

L = 0;

for i1=1:2
    for i2=1:2
        for i3=1:2
            for i4=1:2
                for i5=1:2
                    for i6=1:2
                        T(1) = Mu(i1);
                        T(2) = Mu(i2);
                        T(3) = Mu(i3);
                        T(4) = Mu(i4);
                        T(5) = Mu(i5);
                        T(6) = Mu(i6);

                        pO_T = 1;
                        for j=1:6
                            pO_T = pO_T * gauss1d(O(j), T(j), sigma);
                        end;

                        pT = 1;
                        for j=1:6
                            pT = pT * 0.5;
                        end;

                        L = L + pO_T * pT;
                    end;
                end;
            end;
        end;
    end;
end;

A.2 The GA Algorithm

This program output the best individual of each iteration to MATLAB format, which is used for visualize the GA process in figure 3.

A.2.1 The Source Code in C++

\#include <iostream>

//
// This program is a demo for estimate means of a
// mixture distribution containing 2 Gaussians
//
#include <iostream>
```cpp
#include <vector>
#include <ctime>
#include <cmath>
const double pi = 3.14159265357989;

using namespace std;

// Return float random number in [0,1)
inline double frand()
{
    return (float)rand()/(float)RAND_MAX;
}

// The samples drawn from the mixture distribution
// with known parameters
double O[] = {
    999999, // no use
    1.2083, 0.4473, 0.9893,
    -1.0264, -1.0000, -1.1741
};

const size_t _PopSize = 20;
const size_t _ChromLen = 2;
const float _GeneRangeMin = -2.5f;
const float _GeneRange = 5.0f;
const float _SurviveRatio = 0.1f;
const float _ReproduceRatio = 0.5f;
const float _MutateRatio = 0.5f;
const float _MutateDist = 4.0f;
const int _MaxIterNum = 40;

struct t_Individule
{
    float chromosome[_ChromLen]; // the parameters to be estimated
    float fitness; // the true likehood to given samples

t_Individule()
{
    fitness = 0;
    for ( size_t i=0; i<_ChromLen; i++ )
        chromosome[i] = 0;
}
};

bool better(t_Individule i1, t_Individule i2)
```
{  return i1.fitness > i2.fitness;
}

ostream & operator << (ostream & ostr, const t_Individule & i)
{
    ostr << "fitness: " << i.fitness << '	'
        << "chromosome: ";

    for ( size_t j=0; j<_ChromLen; j++ )
        ostr << i.chromosome[j] <<'	';

    return ostr;
}

typedef vector<t_Individule> t_Population;

void InitPopulation(t_Population & pop, t_Population & buf)
{
    pop.resize(_PopSize);
    buf.resize(_PopSize);

    for ( size_t i=0; i<_PopSize; i++ )
    {
        for ( size_t j=0; j<_ChromLen; j++ )
            pop[i].chromosome[j] =
                (_GeneRangeMin + _GeneRange) * 0.5 +
                (frand()-0.5) * _GeneRange * 0.1;
    }
}

double gauss(double x, double mu, double sigma2)
{
    double k = 1/sqrt(2 * pi * sigma2);
    double e = - (x-mu)*(x-mu) / (2* sigma2);
    double y = k * exp(e);
    return y;
}

double true_likelihood_s(double * O, double * Mu, double sigma)
{
    double * T = new double[7];
double L = 0;

for (int i1=0; i1<2; i1++) {
    for (int i2=0; i2<2; i2++) {
        for (int i3=0; i3<2; i3++) {
            for (int i4=0; i4<2; i4++) {
                for (int i5=0; i5<2; i5++) {
                    for (int i6=0; i6<2; i6++) {
                        T[1] = Mu[i1];
                        T[2] = Mu[i2];
                        T[3] = Mu[i3];
                        T[4] = Mu[i4];
                        T[5] = Mu[i5];
                        T[6] = Mu[i6];

                        double pO_T = 1;
                        for (int j=1; j<=6; j++)
                            pO_T *= gauss(O[j], T[j], sigma);

                        double pT = 1;
                        for (int j=1; j<=6; j++)
                            pT *= 0.5;

                        L += pO_T * pT;
                    }
                }
            }
        }
    }
}

delete [] T;
return L;

void CalcFitness(t_Population & pop)
{
    for (size_t i=0; i<_PopSize; i++)
    {
        double t[2] = {
            pop[i].chromosome[0],
            pop[i].chromosome[1]};

        pop[i].fitness = true_likelihood_s(O, t, 0.3);
    }
}

void SortOnFitness(t_Population & pop)
{ 
    sort(pop.begin(), pop.end(), better);
}

void Mate(t_Individule & kid, 
          t_Individule & p1, t_Individule & p2)
{
    for ( size_t i=0; i<_ChromLen; i++ )
    {
        kid.chromosome[i] =
            (p1.chromosome[i] + p2.chromosome[i]) /2.0f;
    }
}

void Mutate(t_Individule & in)
{
    size_t iPos = rand()%_ChromLen;
    in.chromosome[iPos] += (frand()-0.5) * _MutateDist;

    if ( in.chromosome[iPos] < _GeneRangeMin )
        in.chromosome[iPos] = _GeneRangeMin;
    else if ( in.chromosome[iPos] > _GeneRangeMin + _GeneRange )
        in.chromosome[iPos] = _GeneRangeMin + _GeneRange;
}

int main(int argc, char ** argv)
{
    t_Population a, b;
    t_Population * pop = &a, * buf = &b, * temp = NULL;

    InitPopulation(*pop, *buf);
    cout << "steps = [ \n";

    for ( int iter=0; iter<_MaxIterNum; iter++ )
    {
        CalcFitness(*pop);
        SortOnFitness(*pop);

        cout << (*pop)[0].chromosome[0] << " "
             << (*pop)[0].chromosome[1] << " "
             << (*pop)[0].fitness << endl;

        if ( (*pop)[0].fitness == 0 )
            break;
    }
}
size_t survived = _PopSize*_SurviveRatio;

// Survive
copy(pop->begin(), pop->begin() + survived,
     buf->begin());

// Reproduce
for ( size_t i=survived; i<_PopSize; i++ )
{
    size_t iP1 = rand() % _PopSize*_ReproduceRatio;
    size_t iP2 = rand() % _PopSize*_ReproduceRatio;
    Mate((*buf)[i], (*pop)[iP1], (*pop)[iP2]);

    if ( rand() < RAND_MAX * _MutateRatio )
        Mutate((*buf)[i]);
}

// Swap
    temp = pop;
    pop = buf;
    buf = temp;

cout << "]};
};